Developer
Connection
Recommended Title

APPLESCRIPT

IN A NUTSHELL

A Desktop Quick Reference

O'REILLY®

Bruce W. Perry

Mac

APPLESCRIPT IN A NUTSHELL

AppleScript in a Nutshell is the first comprehensive reference to AppleScript, the popular scripting language that gives both power users and sophisticated scripters the ability to automate repetitive tasks within the Macintosh operating system and built-in applications.

In this concise and well-organized reference, scripters will find the most up-to-date coverage of AppleScript available, including coverage of the version for Mac OS 9.1 and Mac OS X.

The book is divided into the following parts:

Part I provides an introduction to AppleScript and the Script Editor, a free AppleScript development tool that installs with the Macintosh.

Part II provides an extensive core language reference that includes syntax and code examples and detailed descriptions.

Part III is devoted to the scripting of system-level Mac OS 9 and 9.1 programs, such as the Apple System Profiler and the Finder.

Part IV is dedicated to the scripting of the Mac's control panels and extensions.

An appendix is devoted to osaxen (scripting additions), which are powerful language extensions to the built-in AppleScript commands that you can use virtually anywhere in your script.

AppleScript in a Nutshell is an essential desktop reference that puts the full power of this user-friendly scripting language into every AppleScript user's hands.

Developer Connection
Recommended Title

Apple Computer, Inc. boldly combined open source technologies with its own programming efforts to create Mac OS X, one of the most versatile and stable operating systems now available. In the same spirit, Apple has joined forces with O'Reilly & Associates, Inc. to bring you an indispensable collection of technical publications. This logo indicates that the book has been technically reviewed by Apple engineers and is recommended by the Apple Developer Connection.

http://www.apple.com/developer

O'REILLY®
www.oreilly.com

US $34.99 **CAN $34.99**
ISBN: 978-1-565-92841-1

APPLESCRIPT
IN A NUTSHELL

A Desktop Quick Reference

APPLESCRIPT IN A NUTSHELL

A Desktop Quick Reference

Bruce W. Perry

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

AppleScript in a Nutshell
by Bruce W. Perry

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Troy Mott

Production Editor: Catherine Morris

Cover Designer: Ellie Volckhausen

Printing History:

June 2001: First Edition.

ISBN: 978-1-565-92841-1
[LSI] [2013-09-20]

Table of Contents

Part IV: Scripting Mac OS 9 Control Panels and Extensions

Part VI: Appendixes

Preface

AppleScript continues to evolve on Mac OS 9 and Mac OS X as the ultimate scripting tool for the Macintosh. AppleScript's power to automate the operating system and complex applications such as graphics, desktop-publishing, and database programs, as well as a friendly English language dialect that helps ambitious scripters get up to speed quickly with their own applets, is not matched by any other platform's programming language. Yet, only a small percentage of Macintosh users are even aware that AppleScript is installed with their operating system. Those who are aware of AppleScript's presence on their machine often do not take full advantage of this tool to automate their daily computing activities, both on their local machine and over the Internet.

Who should and can use AppleScript? The following users come to mind right away: system administrators who are automating tasks with networks and applications; web and graphics professionals who want to control the development of web sites and publications; scientists, mathematicians, and engineers who require applets to make calculations and automate their own software tools, as well as day-to-day programmers and students who are designing and prototyping new programs. Not to mention everday users who want to automate their own computing tasks, such as file and folder backups.

If you are on a Macintosh, then you should be putting AppleScript to work for you.

The purpose of this book is primarily three-fold:

1. Describe AppleScript and its tools (Part I) and provide a core language reference (Part II) that all users can keep next to their computers as they write new scripts.
2. Provide detailed descriptions, examples, and reference information on how to script the numerous system-level programs on Mac OS 9 (Parts III and IV) and Mac OS X (Part V), such as the Finder on both OS versions, Sherlock, and Network Setup Scripting.

3. Give scripters general insight on how to approach the scripting of several programs that can be automated by AppleScript, such as Adobe Illustrator and Photoshop, FileMaker Pro, QuarkXPress, SoundJam MP, and OutLook Express. The mantra is, study the "application class" in the program's AppleScript dictionary and you'll be up and running with scripting that program before you know it. (Chapter 1 discusses the application class in general terms; while the application classes of all the various system components are described in detail throughout the book.)

Hopefully, this book will help reveal AppleScript to more Macintosh users, thus providing them with another outlet for creativity and productivity.

Organization of This Book

AppleScript in a Nutshell is structured in six parts.

Part I, *Introduction to AppleScript*

This section provides an overview of AppleScript and Script Editor, the free AppleScript development tool that installs with the Macintosh. Quick studies and experienced programmers will probably be able to develop their first AppleScripts (if you have never used AppleScript before) based on a reading of this introductory section alone. Chapter 1 describes how AppleScript is primarily used and also describes the relevance to AppleScript of *Apple events*, an internal messaging system that the Macintosh operating system uses for interapplication communication. The end of Chapter 1 summarizes AppleScript's core language features (Part II provides a more comprehensive language reference). You can use Chapter 2 as a helpful reference to Script Editor as you use this Apple Computer tool to develop your scripts.

Chapter 1, *AppleScript: An Introduction*

This AppleScript overview includes a description of how AppleScript is primarily used, an Apple-event tutorial, and a compressed language reference for those who want to dive right into scripting. Novice users should start here with the book, while very experienced AppleScripters may use this section as a review or skip over it.

Chapter 2, *Using Script Editor with OS 9 and OS X*

This chapter describes all of Script Editor's primary menu commands and controls. It also explains the various options for saving AppleScript files.

Part II, *AppleScript Language Reference*

If scripters need more information on specific language features, this is the place to look. The core-language information is presented with syntax examples, code examples, and text descriptions. Everything is arranged in alphabetical order to make things easy to locate. This includes the various data types (i.e., how AppleScript stores data in memory), operators (such as the common Math operators and

the string-concatenation operator "&"), and how to set AppleScript variables and create user-defined functions, as well as advanced features, such as creating object-oriented script objects (Chapter 9).

Chapter 3, *Data Types*

This chapter describes the built-in AppleScript data types, including `string`, `integer`, `real`, `list`, and `record`. Comparisons with programming languages are made where it is appropriate (e.g., a `list` is like an array, and a `record` is an associative array).

Chapter 4, *Operators*

Use this chapter as a reference to the built-in symbols (e.g., &, +, *, -) that you can use in AppleScript expressions.

Chapter 5, *Reference Forms*

AppleScript provides several English-language terms to use when the script refers to objects on your computer system, such as files, folders, disks, and applications. This chapter is an alphabetical reference to these terms (e.g., `first`, `every`, `id`, `where`).

Chapter 6, *Variables and Constants*

AppleScript, like other languages, uses variables as placeholders that represent data (e.g., strings or numbers). This chapter describes the rules for naming and creating your own variables; it also provides a reference to AppleScript's constants and predefined variables (like `pi`).

Chapter 7, *Flow-Control Statements*

This chapter is an alphabetical reference to AppleScript's flow-control statements, such as `if`, `repeat`, `try`, `exit`, and `continue`.

Chapter 8, *Subroutines*

This chapter is a tutorial on creating user-defined subroutines, which are also called handlers, functions, or methods (in object-oriented parlance). The second part of this chapter describes five special handlers in AppleScript: *idle*, *open*, *quit*, *reopen*, and *run*.

Chapter 9, *Script Objects and Libraries*

AppleScripters can create script objects, which are user-defined types that can have their own attributes and methods. This chapter also describes function libraries, which are script objects that give other external scripts the ability to load and/or call the object's own functions.

Part III, *Scripting Mac OS 9 Applications*

This section is devoted to the scripting of system-level Mac OS 9 programs, such as Apple System Profiler, Keychain Scripting, the Finder, Network Setup Scripting, and Sherlock 2. The scriptable control panels and extensions are covered in the next section, Part IV. The programs that are covered in this section for the most part have comprehensive AppleScript dictionaries and can be used to extend your computer's capabilities (particularly with AppleScript!); however; they are not control panels or extensions. The exception to this scheme is Apple Guide, which is an extension but was included in this section so that the reader has access in a single chapter to a description of AppleScript and the help-related programs. Each chapter describes the purpose of the application, then describes each dictionary command and class in a reference-style form.

Chapter 10, *Apple Guide and Help Viewer*

This chapter describes the dictionaries and includes scripting tips for Apple Guide, the traditional automated Apple-help program, and the newer browser-based Help Viewer tool.

Chapter 11, *Apple System Profiler*

Accessible from the Apple menu, Apple System Profiler displays a wealth of information about the hardware and software on your system. This chapter describes its commands and classes and includes numerous code examples.

Chapter 12, *Keychain Scripting and Apple Verifier*

These are two Apple-security tools. Keychain Scripting is used to encrypt files and passwords, and Apple Verifier can verify digitally-signed files. This chapter tells where to find these applications and describes their commands and classes in reference form.

Chapter 13, *Desktop Printer Manager*

Scripters can use Desktop Printer Manager, a program introduced with Mac OS 8.5, to create and manage desktop icons that can be used for printing or otherwise processing documents and files. This chapter describes the proper syntax for controlling this application with AppleScript and also includes a reference to its dictionary commands and classes.

Chapter 14, *Mac OS 9 Finder Commands (MAC OS 9)*

The Finder is the Mac OS 9 application that controls the user's visual interface to the computer: its desktop controls as well as hard disks, network volumes, printers, and other devices. A lot of fun and useful AppleScripts deal with automating Finder activities, such as reading from and writing to files. This chapter covers the Finder commands, like *restart*, *shutdown*, *sleep*, and *make*, with detailed references to each command and any of their parameters.

Chapter 15, *Mac OS 9 Finder Classes*

This chapter covers the Finder classes, which are all the objects or things you are likely to control when scripting the Finder (e.g., files, folders, disks, and running applications). *Finder Classes* provides a detailed reference to each object's elements (if any) and properties.

Chapter 16, *Network Setup Scripting*

As the Macintosh becomes a sophisticated client and server on TCP/IP networks, *Network Setup Scripting* shows how you can use the commands and classes of this program with Open Transport to script a machine's various network configurations.

Chapter 17, *Scripting Sherlock 2*

You can automate sophisticated searches of local networks and the Web with AppleScript and Sherlock 2. *Scripting Sherlock 2* provides a description of this program and a reference, with code examples, to its commands (e.g., *index containers*, *search*) and classes.

Chapter 18, *URL Access Scripting*

URL Access Scripting describes the *download* and *upload* commands of this program, which can be used with the FTP and HTTP protocols to grab and save files off the Web.

Part IV, *Scripting Mac OS 9 Control Panels and Extensions*

This section is dedicated to the scripting of the Mac's control panels and extensions, which are located in the Control Panels and Extensions folders of the System Folder. Each chapter describes the purpose of this system software, then includes a reference to their dictionary commands and classes. Some of the more exciting new scriptable technologies are included in this section, including Apple Data Detectors, Folder Actions, and the Speech-related extensions in Chapter 30.

Chapter 19, *Appearance Control Panel*

This scriptable control panel lets you use AppleScript to set and change the visual and audible aspects of your computer, such as its background color, the font for desktop text, and how window title bars and scroll bars work. We show you how to do this and include a detailed reference to this software's commands and classes.

Chapter 20, *Apple Data Detectors Extension*

This chapter describes a powerful scripting technology by which you can assign an AppleScript to be triggered based on certain information that a user selects inside of a contextual menu, such as an email or web address. *Apple Data Detectors Extension* describes the Apple Data Detectors scripting-addition class and commands in reference form.

Chapter 21, *Apple Menu Options Control Panel*

This chapter describes how to use AppleScript to automate various menu items (e.g., Recent applications, documents, and servers) in the Apple menu (the drop-down menu in the upper-left part of the computer screen).

Chapter 22, *Application Switcher Extension*

The Application Switcher is the floating palette that the user can "tear" off of the Application menu (on the upper-right part of the computer screen). This chapter describes how to set various Switcher elements (e.g., its size, position, button order) with AppleScript and includes a reference to its extensive `application` class.

Chapter 23, *ColorSync Extension*

ColorSync Extension describes the AppleScript commands and classes for this built-in Macintosh software, which helps synchronize color-matching between the devices that create an image (e.g., scanners) and printers.

Chapter 24, *File Exchange Control Panel*

This chapter describes the File Exchange commands that you can use to create new extension mappings (i.e., a way to tell the Macintosh how to handle files with certain extensions like *.html*), for instance, or view the existing file-type mappings on a machine.

Chapter 25, *File Sharing Control Panel*

This chapter first summarizes file sharing on the Macintosh, which establishes the level of access network users have to a machine's disks and folders. Then it shows how to create new users or groups (or delete miscreants) with code examples and a reference section on File Sharing's dictionary commands and classes.

Chapter 26, *Folder Actions Extension*

Folder actions are AppleScripts that are triggered when items are added to or removed from a folder. Folder action commands constitute the Folder Actions suite of the Standard Additions osax and the dictionary commands that derive from the Folder Actions extension. This chapter describes both sets of commands.

Chapter 27, *FontSync Control Panel and Extension*

This chapter describes the dictionaries for the FontSync control panel and extension. They are used to synchronize the fonts between devices during image production and printing.

Chapter 28, *Location Manager Control Panel*

This chapter shows how you can use AppleScript to switch between the various computer and networking configurations that are displayed by the Location Manager control panel.

Chapter 29, *Memory and Mouse Control Panels*

This chapter describes the dictionary commands and classes for both the Memory and Mouse control panels. For example, the chapter shows how you can use an applet to find out about the computer's virtual-memory settings or disk-cache size.

Chapter 30, *Speech Listener and SpeakableItems Extension*

This chapter describes the different ways that you can integrate speech into your scripts, such as the *listen for* and *say* AppleScript commands. Speech listener is actually an application that is located in the Scripting Additions folder of the System Folder, but it will not work unless the Speech Recognition extension is installed and enabled.

Chapter 31, *Web Sharing Control Panel*

This chapter describes the functionality of the Web Sharing control panel and also gives an example of how to use AppleScript with a Common Gateway Interface (CGI) script. CGI scripts execute in response to web page requests, in order to process the incoming data from a form a web user has filled out, for instance. The Web Sharing control panel can be used to allow a computer to perform as a light-weight web server.

Part V, *Scripting the Mac OS X System*

AppleScript is in a state of flux and evolution on the new Mac OS X system. AppleScript also faces tremendous competition from the programming tools that come with (and can be installed on) Mac OS X, such as shell scripting tools, Perl, and Java. Nevertheless, this section will describe what you can do with Apple-Script and three Mac OS X programs that *can* be used with AppleScript: Mail, Terminal application (a command-line tool), and TextEdit. Part V begins with a discussion of AppleScript and scripting the new Mac OS X Finder, which is the OS 9 Finder after a major facelift.

Chapter 32, *Scripting the OS X Desktop*

This chapter explains some of the familiar Finder-like scripting that you can accomplish on Mac OS X, such as getting information about desktop items (e.g., files, folders, and disks) and making new files. This chapter compares the Mac OS X Finder dictionary to the Mac OS 9 Finder dictionary (and finds few differences, but that is likely to change with new OS X versions).

Chapter 33, *Scripting Mail*

This chapter describes the use of AppleScript with Apple Computer's new email application, aptly called "Mail." This chapter provides descriptions and code examples on setting up a new mail message and getting information about an email account.

Chapter 34, *Executing Scripts with the Terminal App*

Terminal application is the command-line tool or interface (a window or shell that you type script commands into) that comes with Mac OS X. This chapter shows how you can create, compile, and execute AppleScripts from the Terminal program.

Chapter 35, *Scripting TextEdit*

It is likely that the TextEdit's available AppleScript commands will change with new Mac OS X releases, so this chapter focuses on TextEdit's major commands (e.g., *count*, *open*, *save*) and text-related classes, such as `character`, `document`, `paragraph`, and `text`.

Part VI, *Appendixes*

Our AppleScript book would not be complete without a description and reference information on the many scripting additions or "osaxen" that veteran scripters use in almost every script (remember *display dialog* or *current date*?). Appendix A covers the Standard Additions (a group of scripting additions that Apple Computer bundles with the OS installation) that are installed with both Mac OS 9 and Mac OS X. This section describes each of the Standard Additions (e.g., *ASCII number*, *beep*, *choose application*) and any parameters that these osax commands use. Appendix B, *AppleScript Resources*, is a list of URLs that are relevant to AppleScript users.

Appendix A, *Standard Scripting Additions*

This appendix focuses on the several dozen Standard Addition scripting additions, which are installed along with Mac OS 9 and Mac OS X. These are extensions to the built-in AppleScript commands that you can use virtually anywhere in your script (Chapter 1 also discusses scripting additions). The Standard Additions are located in the *startup disk:System Folder:Scripting Additions* folder in OS 9 and, with Mac OS X, */System/Library/ScriptingAdditions/* (the primary location on OS X).

Appendix B, *AppleScript Resources*

This is an extensive list of web pages relating to Macintosh scripting and AppleScript.

Conventions Used in This Book

The following typographical conventions are used in this book:

`Constant width`
: Is used to indicate command-line computer output and code examples, as well as AppleScript class names, objects, parameters, data types, properties, methods, constants, variables, and flow-control statements like `repeat`.

`Constant width bold`

Is used to indicate user input in examples.

Italic

Is used to introduce new terms and to indicate URLs, user-defined files and directories, commands, file extensions, filenames, directory or folder names, and UNC pathnames.

Italic is also used to highlight chapter titles and, in some instances, to visually separate the topic of a list.

This is an example of a note, which signifies valuable and timesaving information.

This is an example of a warning, which alerts to a potential pitfall in the program. Warnings can also refer to a procedure that might be dangerous if not carried out in a specific way.

Keyboard Shortcuts

When keyboard shortcuts are shown (*Command-N*), a hyphen means that the keys must be held down simultaneously, while a plus means that the keys should be pressed sequentially.

Path Notation

We use a shorthand path notation to show you how to reach a given user interface element or option. The path notation is relative to a well-known location. For example, the following path:

Script Editor's File → Open Dictionary

means "Open the Script Editor's File menu, then choose Open Dictionary."

File path delimiters

AppleScript uses the colon to separate the directories in a file path, as in *MyStartupDisk:Desktop Folder:myfile*. The major scripting additions that deal with file paths, such as *choose file*, *choose file name* (Mac OS X and OS 9.1), *choose folder*, and *path to*, display their file paths in `alias` return values as colons. The chapters that deal with Mac OS X, however, will often identify the locations of files and folders with the Unix-style slash character `/` as the path delimiter (e.g., */users/bruceper/documents/*). This is the path delimiter used by Darwin, which is the core operating system for Mac OS X and has Unix origins. The opening slash character in the file path */users/bruceper/* sets the beginning of the path to the "users" folder on the disk or partition where Mac OS X is located. AppleScript on Mac OS X still generally uses colons as the

path delimiter, however, which maintains consistency with older scripts (OS 8/9). One place where you can use the slash character to locate a path for AppleScript is in setting the `target` property for a Finder window, as in:

```
set the target of Finder window 1 to "/users/bruceper/"
```

Italic `Constant Width`
: On occasion, you will find a command description such as *connect* `remote access configuration` object, which means that the *connect* command takes a `remote access configuration` object as a parameter.

How to Contact Us

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
1-800-998-9938 (in the U.S. or Canada)
1-707-829-0515 (international/local)
1-707-829-0104 (FAX)

You can also send us messages electronically. To be put on the mailing list or request a catalog, send email to:

info@oreilly.com

To ask technical questions or comment on the book, send email to:

bookquestions@oreilly.com

We have a web site for the book, where we'll list examples, errata, and any plans for future editions. You can access this page at:

http://www.oreilly.com/catalog/aplscptian/

For more information about this book and others, see the O'Reilly web site:

http://www.oreilly.com

Acknowledgments

Every book is a prodigious effort that could never be accomplished by the author alone. I would first like to thank my wife Stacy LeBaron and daughter Rachel, who have patiently and sympathetically waited for me to emerge from what has seemed, to them, a never-ending process of word- and code-crunching. Next I would like to gratefully acknowledge Anne and Robert Perry, my parents, who have instilled in me a love of books and the intellectual discipline it takes to digest and write them. The O'Reilly team has been indispensable: my editors Simon Hayes, for his insightful nudging and prodding when I first proposed the project, and the tireless efforts of Troy Mott and Bob Herbstman as the book entered the final production stages.

Chris Stone at O'Reilly also has made tremendous contributions to the shaping of this book. Thanks to Bill Cheeseman and Paul Berkowitz for helpful technical reviews of several chapters. Finally, I would also like to acknowledge all the AppleScript experts and engineers at Apple Computer who took time out from their busy schedules to comment on this book.

PART I

Introduction to AppleScript

CHAPTER 1

AppleScript: An Introduction

AppleScript is a scripting tool that installs with the Mac OS, including the newest release, Mac OS X. Programmers and power users use AppleScript to create scripts and applets, which are small Mac programs that can both accomplish useful tasks on their own and greatly extend the capabilities of other software systems.

This chapter covers the following topics:

- How AppleScript is used (for example, for software automation and the attaching of scripts within an application's menus).
- An overview of Apple events, a messaging technology that AppleScript uses to control scriptable applications. This section briefly describes (1) how AppleScript code sends Apple events, as well as (2) Apple event classes and objects.
- Two applications that you can use to access and run your scripts from the file system: Script Runner (for Mac OS X) and OSA Menu (Mac OS 9). Chapter 2, *Using Script Editor with OS 9 and OS X*, is completely devoted to Script Editor, which is the script development environment that installs with the Macintosh OS.
- AppleScript's language elements, such as data types, variables, handlers (i.e., subroutines or functions), and flow-control statements. This is a "quick reference" for the readers who want to dispense with narrative and dive right into scripting. Part II then covers all of these elements in detail.

How Is AppleScript Used?

AppleScript can be used for both simple, self-contained solutions, such as a program whose sole purpose is to monitor how much space is left on a disk, and comprehensive systems that automate or control a suite of software programs. Let's begin with a simple script type, a standalone applet that is not attached to or designed to automate another software program.

You generally create an applet by typing AppleScript source code into an Apple Computer scripting program called Script Editor. You then compile the script (if it does not have any errors) into a small program called a compiled script or an applet that can be double-clicked on the desktop. An AppleScript applet is a self-contained program with its own desktop icon, while a compiled script requires a host program like Script Editor or Script Runner (see "Using Script Runner with OS X" later in this chapter) to run it. Figure 1-1 shows an applet icon. Chapter 2 also explains the various options for saving an AppleScript.

Figure 1-1: An applet icon

AppleScript is a great tool for writing everyday utilities, such as managing files, folders, disks, and networking activities. The utility scripts provide all the functionality you need, without the necessity to automate another software program. These tasks, such as file backups or getting a browser to access certain web pages, would be time-consuming and tedious if they always had to be performed manually. Two examples of scripts that I run at least once every day are:

- A script that displays a dialog listing the names of all of the running programs on the Mac, including invisible background processes. I can select one or more of these programs and click a button on the dialog window to close them.
- An applet that calculates the remaining free space on all of the volumes that are mounted on the desktop, then displays the result for each volume and the total free storage space on all of the volumes put together.

A single hard disk can be divided into several volumes, which the Mac OS represents as disk icons on the user's desktop.

By now you would probably like to see just what applet source code looks like. The script in Example 1-1 displays the largest unused block of Random Access Memory (RAM) remaining on the computer where the script is run.

Example 1-1: AppleScript Displaying the Largest Block of Free Memory

```
tell application "Finder"
   activate
   set memblock to (largest free block / 1024 / 1024)
display dialog "The largest free block is now about " & (memblock) & ¬
" megabytes."
end tell
```

This script asks the Finder application for a piece of data that the Finder maintains called "largest free block." This represents the size of the largest free memory block in bytes. The following script fragment:

```
(largest free block / 1024 / 1024)
```

divides this byte-size figure twice by 1024 to represent the result in megabytes, since most people convey the amount of computer memory they have using this measurement. *display dialog* is an often-used extension to the built-in AppleScript language called a scripting addition, which I explain later in this chapter (Appendix A, *Standard Scripting Additions*, of the book is devoted to descriptions of the standard scripting additions that are installed with Mac OS 9 and OS X). *display dialog* shows a dialog window containing the message label that you specify in the source code following the *display dialog* command, as in this part of Example 1-1:

```
display dialog "The largest free block is now about " & (memblock) & ¬
" megabytes."
```

The `tell` statement that opens the script, such as:

```
tell application "Finder"
```

is AppleScript's method of targeting an application to request some data from it or to control the program in some manner. Since the script displays some Finder information to the computer user, the *activate* command is used to make the Finder the frontmost program (i.e., its windows, if any are open, become the active desktop windows). `tell` statements, commands, and other syntax elements are described elsewhere in this chapter, as well as in detail in Chapters 3 through 8.

Automation

Along with creating a number of useful utilities, AppleScript has won a reputation as a premier tool for automating software workflows. In workflows, one or more separate software programs cooperate in a sequence of actions to complete a job. This means that launching an AppleScript can orchestrate several actions that involve software applications that are not otherwise designed to share data or call each other's menu commands. AppleScript does the calling of each program's commands (targeting them in a similar manner to how the Finder is targeted in Example 1-1), acting as a conductor for busy software medleys. AppleScript has earned the undying loyalty of many Mac scripters in the print and web publishing industries by its ability to simultaneously control applications such as QuarkX-Press, Adobe Illustrator, InDesign, and Photoshop, Canto Cumulus, FileMaker Pro, as well as the Microsoft Office members like Word and Excel.

As an example of automation, I designed an AppleScript in the summer of 2000 to convert thousands of text files to web pages. A company that publishes legal decisions wanted to make them available to a search engine on their web site. Since they were already plain text or Word files, and the page designs were very simple, we used an AppleScript to feed the pages to Word and to trigger its "Save as HTML…" menu command (which creates a simple, almost crude, web-page design at best). The company converted about 20,000 legal decisions in a matter of days, using this rather modest script that I developed in Script Editor.

Apple Computer has traditionally urged Mac software developers to make their programs "scriptable," and thus increase the market and following for those programs. It usually does. For example, Illustrator and Photoshop* are generally much more scriptable on the Mac platform than their Windows versions, which may influence some buyers to prefer the Mac versions (along with the fact that graphics professionals tend to prefer the Apple platform).

Similar to OLE automation on the Windows platform, many software programs allow themselves to be automated by exposing an object model to a scripting tool, in this case AppleScript. Conceptually, an object model is a tree-like diagram (see Figure 1-7 later in this chapter) showing the objects or "things" that the software represents as computer data (such as files, folders, or database records), as well as the objects' attributes or properties. AppleScript talks to these scriptable programs by exchanging *Apple events* with them. These are high-level operating-system events that are used for interapplication communication on the Mac. See the section "Apple Events" for more information.

With the release of Mac OS 9 and OS 9.1 in 2000 and early 2001, Apple Computer has made most of the computer's built-in software controllable by AppleScript. These are some of the scriptable OS 9 applications and control panels:

- Appearance control panel
- Apple Help Viewer
- Apple System Profiler
- Apple Verifier
- Application Switcher
- ColorSync extension
- File Exchange
- Memory control panel
- Sherlock 2 (the Find application)
- Speech Recognition

Some previously scriptable features have not been included in the Mac OS X installation, including preferences scripting, Folder Actions, printing scripting, and program linking. Future OS X releases will address these elements, according to Apple Computer, which adds that a number of Mac OS X applications are scriptable (with some qualifications):

- Finder (some Finder commands, such as *move* or *duplicate*, are not yet implemented or are not yet functioning)
- TextEdit
- Mail
- Sherlock

---

* Photoshop requires the licensing of the PhotoScripter plugin from Main Event Software (*http://www.mainevent.com*) to be extensively automated with AppleScript.

- QuickTime Player (the Pro version of this application is quite scriptable; visit *http://www.apple.com/applescript* to download the QuickTime scripts collection for Mac OS X)
- Apple System Profiler
- Stuffit Expander
- Internet Explorer
- ColorSync Scripting
- URL Access Scripting
- Image Capture Extension (a background application that works with the Image Capture program; its dictionary supports the scaling and rotating of image files)

In addition, the AppleScript engineers are apparently working on ways to let AppleScript interact with the command-line shells that come with OS X, such as the Bourne shell. OS X already permits the launching of AppleScripts from a shell (see Chapter 34, *Executing Scripts with the Terminal App*).

Attachability/Recordability

If an application is either attachable or recordable (or both), it is considered a near paragon of scriptability. Attachable means that you can create a script and then attach it to a program, so that the script is added to the program's internal menus. Applications usually implement attachability with Mac OS 9 by providing a folder for scripts and a menu item on their menubars that lists these available scripts. Figure 1-2 shows a menubar that contains a list of attached scripts for the BBEdit text editor.

Attached scripts will often run much faster than scripts that run as self-contained applets even if the script doesn't have anything to do with the application it is attached to (i.e., the script never sends Apple events to the host application). For example, I have an AppleScript that reads large web logs (more than 1 MB in size) looking for and recording for later display certain file paths. When attached to BBEdit 5.1, the script runs about six times as fast as it does when run as an applet outside of BBEdit (40 seconds as opposed to about 240 seconds). Try it with some of your own scripts.

A few applications allow themselves to be recorded by Script Editor, which is a great way to get started with scripting them. To do this, open Script Editor and click its Record button (see Chapter 2). You then activate an application and perform the actions that you are trying to record, or simply go in and manipulate its menus to see what happens. Once you click Stop in Script Editor, the Script Editor window will display the AppleScript source code representing the recorded actions. If the application is not recordable, the Script Editor window will be empty after you click Stop. Otherwise, you can then save the AppleScript as a macro that you can use and/or modify in the future. The Finder, BBEdit, and Microsoft Word are examples of recordable applications.

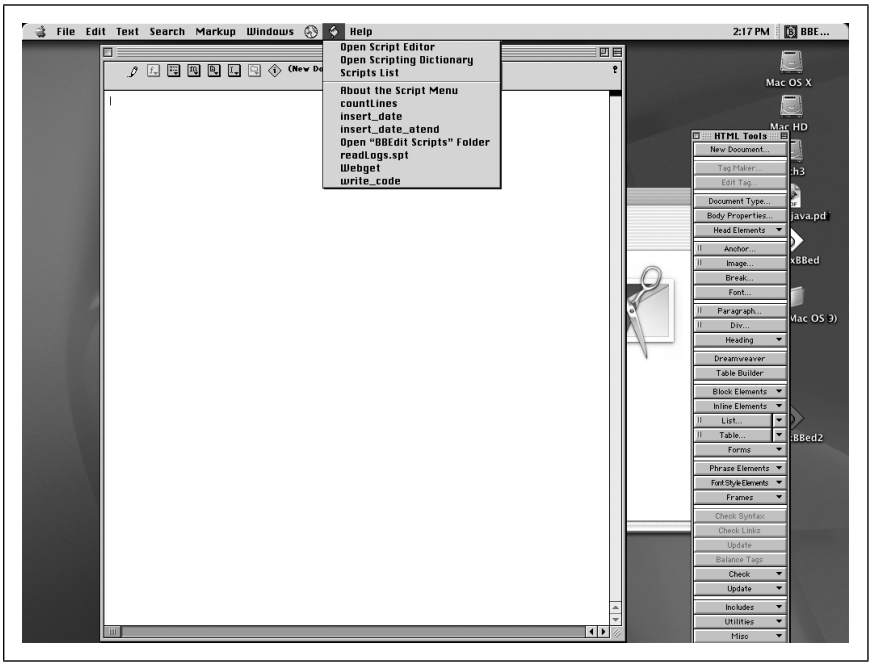

Figure 1-2: Attached scripts in BBEdit

Scripting Additions

Scripting additions are extensions to the AppleScript language. The Standard Additions and some others are written by Apple; however, there are hundreds of scripting additions the scripters can add themselves. They are added by placing the scripting addition file into the Scripting Additions folder. Once installed, scripting additions can be used by any script.

Another term for scripting addition is osax (the plural form is osaxen), which stands for Open Scripting Architecture eXtension. The OSA is explained in an upcoming section of this chapter.

In Mac OS 9, the scripting addition files are stored in the *startup disk:System Folder:Scripting Additions* folder. They are stored in more than one location in Mac OS X, including */System/Library/ScriptingAdditions/*. Examples of two scripitng addition commands that are often used are *display dialog* (see Example 1-1) and *current date*. The latter command returns a `date` object that contains data about today's date and time. The Standard Additions are installed with the Mac OS.

There is a large database of scripting additions at *http://osaxen.com*.

Apple Events

AppleScript and scriptable programs communicate with each other via Apple events or internal, invisible messages. This section provides an overview of how Apple events are implemented with AppleScript. As this information goes beyond basic script development, some readers may choose to jump ahead to the book's language-reference sections, do some scripting, and then revisit this section at another time.

OSA

The Open Scripting Architecture, which has been present on the Mac since the early 1990s, is Apple Computer's mechanism for making interapplication communication available to a spectrum of scripting languages. AppleScript is the OSA language that Apple provides, but there are other OSA-compliant languages, including UserLand Frontier and JavaScript.*

OSA accomplishes this "the-more-the-merrier" approach to scripting systems by using Apple events as the unifying technology. The situation is similar to Open Database Connectivity (ODBC) on the Windows platform, where any client application can talk to different database management systems using ODBC as a common conduit (as long as ODBC driver software exists for that particular database). In terms of OSA, the conduit (on Mac OS 9) is a scripting component that can convert whatever scripting language is used (AppleScript or JavaScript) into one or more properly constructed Apple events. Figure 1-3 shows the same Apple event being sent to an application in two different scripting languages.

Before AppleScripts or other scripting languages can be compiled and run, their corresponding extension files have to be installed (the AppleScript extension is included in an OS 9 installation) and then loaded into the computer's memory. The AppleScript extension or component is depicted in Figure 1-3.

Apple Event Registry

Along with scripting components, another important OSA element is the Apple Event Registry. The Registry is an Apple Computer-maintained database that maps all of the Apple events that the Mac OS standard software uses to a corresponding English-language command. This means that the *activate* AppleScript command is mapped to an *activate* Apple event, *quit* is mapped to a *quit* Apple event, and so on. You can use the Registry to discover the Apple event codes that are used by the Mac's standard software (such as the Appearance control panel, the ColorSync extension, or Sherlock 2). The section "Inside an Apple Event" describes what these codes are.

* Late night Software, Ltd.'s JavaScript for OSA tool, is accessible from *http://www.latenightsw.com*.

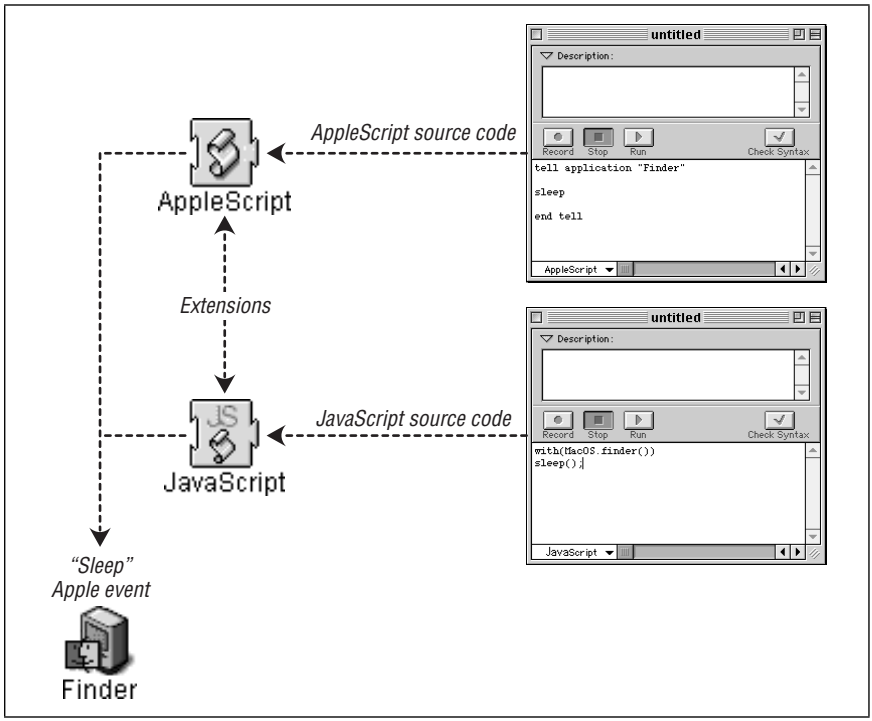

Figure 1-3: OSA scripting tools send Apple events

The AppleScript software development kit (SDK) includes a FileMaker Pro file that contains the Apple Event Registry for AppleScript Version 1.3.4. Go to *http://developer.apple.com/sdk* for more SDK information.

To make them easier to understand and incorporate into applications, Apple events are logically grouped into suites or categories, such as the Database Suite, the Standard Suite, and the Text Suite. All Mac applications are required to support four Standard Suite events (*open*, *print*, *quit*, and *run*; this was the "Required Suite" prior to AppleScript 1.3). This does not mean that all Mac programs do support these events; software developers don't go to jail if they have not implemented these Apple events in their programs. However, this does mean that the vast majority of applications will reliably quit if, for example, your script sends them a *quit* Apple event.

Applications and scripting additions can (and usually do) define their own Apple events and corresponding human-language commands. For example, the BBEdit text editor supports a subset of the Standard Suite of Apple events that you can look up in the Registry database. BBEdit also contains a set of events and classes known as the BBEdit Suite, which is unique to BBEdit. Table 1-1 shows the Standard Suite Apple events and Apple event classes that BBEdit 5.1 supports. It also

shows the Apple events and Apple event classes that are listed in the BBEdit Suite. (The section "Apple Event Classes and Objects" describes Apple event classes in more detail.)

Table 1-1: BBEdit 5.1's Standard Suite and BBEdit Suite

BBEdit Standard Suite Events	*BBEdit Standard Suite Classes*	*BBEdit Suite Events*	*BBEdit Suite Classes*
close	application	insert text	character
count	window	insert file	word
delete	document	insert folder	line
get	Recent file	insert project	text
make		find	text item
revert		replace	selection-object
save		find differences	hit
set		go to line	
		go to function	
		go to marker	
		select current paragraph	
		twiddle	
		change case	
		shift	
		hard wrap	
		insert line breaks	
		remove line breaks	
		unwrap	
		zap gremlins	
		entab	
		detab	
		insert glossary entry	
		get FTP file	
		put FTP file	

Client/Server

The application or applet that initiates an exchange of Apple events is called the client application. The client requests the help of the server ("do something for me!"). The client's Apple event(s) may request data (e.g., text, database records) or just a sequence of actions that the server should take ("Open a file and send me the paragraph that begins with 'Top-secret information.'"). The client can also be thought of as the Apple event "source," and the server can be thought of as a "target." An application can be both an Apple-event client and a target (if a client receives a reply Apple event, then it's the target of that event).

A machine can send up to about 2,000 Apple events per second (and can be as pokey as about 5 per second). This speed depends on factors such as how quickly the target application can process the Apple event(s).

How Many Apple Events Can Your Machine Send?

An Apple Computer engineer suggests that you use the following code to test how many Apple events a particular machine can send per second:

```
set start_time to current date
repeat 1000 times
   tell application "Finder"
      name -- gets the Finder's name, "Finder"
   end tell
end repeat
set elapsed_time to (current date) - start_time
display dialog "Average " & 1000 / elapsed_time & " events per ¬
second"
```

This code sends the Finder 1,000 Apple events, and then displays the event-per-second results. Running this as a compiled script out of Script Editor, my machine (a PowerMac 8500 upgraded to a G3 with plenty of memory) registered only 5 per second. However, when saved as an applet and attached to BBEdit, the speed improvement was 20-fold—100 Apple events per second!

Let's drill down further into Apple events. The upcoming section "Inside an Apple Event" shows you what an Apple event looks like at the system level, using the *sleep* Apple event, a Finder command, as an example.

Inside an Apple Event

Here's how it works when Script Editor compiles and executes the following code, which comprises a complete compilable script:

```
tell application "Finder" to sleep
```

This is what happens:

1. The AppleScript component has to find out which Apple event lies behind the *sleep* command. The component knows that the Finder is one of the places it should look for these details, because the Finder is targeted by the `tell` statement:

   ```
   tell application "Finder"...
   ```

2. Remember that sleep is an English-language term for putting the computer to sleep, but it is implemented as the *sleep* Apple event beneath the surface. Figure 1-4 shows the structure of the sleep Apple event.

   The AppleScript component discovers the attributes of the *sleep* Apple event (e.g., the event id) from a segment of the Finder file called the Apple event terminology extension (`'aete'`) resource. The `'aete'` resource maps the sleep script command to the Apple event depicted in Figure 1-4.

3. The component then sends that Apple event to the Finder, which responds to sleep by powering down the computer.

Here is an explanation of the structure behind the Apple event in Figure 1-4.

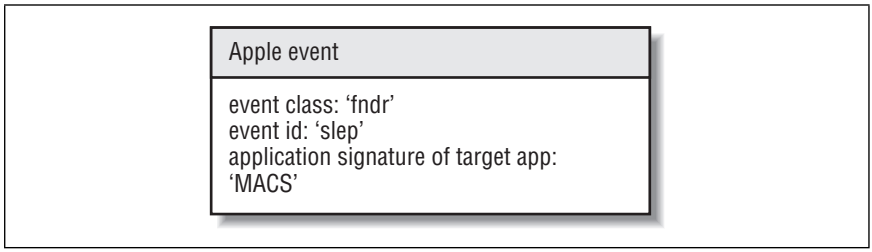

Figure 1-4: A sleep Apple event

Every Apple event is comprised of unique four-character codes that represent the:

- Event class
- Event id
- Address of the target application

The event class represents a grouping of similar Apple events. The event id uniquely identifies the Apple event. The target address is a complex data structure that could contain the application's creator code or its Process Serial Number (PSN) or another piece of identifying information. For example, the *sleep* Apple event has event class `'fndr'` and event id `'slep'`. Table 1-2 contains the event classes and event ids for the Standard Suite in the Apple Event Registry. Apple events often get reorganized within different suites when Apple updates its Registry.

Table 1-2: Apple Event Codes for Standard Suite

Event	*Event Class*	*Event Id*	*Class*	*Class Id*
`open`	`aevt`	`odoc`	`application`	`capp`
`run`	`aevt`	`oapp`	`document`	`docu`
`reopen`	`aevt`	`rapp`	`file`	`file`
`print`	`aevt`	`pdoc`	`alias`	`alis`
`quit`	`aevt`	`quit`	`selection-object`	`csel`
`close`	`core`	`clos`	`window`	`cwin`
`count`	`core`	`cnte`	`insertion-point`	`cins`
`delete`	`core`	`delo`		
`duplicate`	`core`	`clon`		
`exist`	`core`	`doex`		
`make`	`core`	`crel`		
`move`	`core`	`move`		
`save`	`core`	`save`		

Most of the time, however, a scripter does not have to deal with event classes and event ids, just their AppleScript language equivalents.

Apple events specify the target programs that should receive the Apple event. Otherwise, your script would cause an execution or runtime error, because the operating system does not know where the Apple event is supposed to go.

The common way to specify the target programs for an Apple event in AppleScript is to use a code such as in Example 1-2. You enclose the Apple events you will send to a program within the tell block, as in Example 1-2, which sends a quit Apple event to "FileMaker Pro".

Example 1-2: A Script Targeting FileMaker Pro

```
tell application  "FileMaker Pro"
   quit
end tell
```

The value of the application signature attribute in Figure 1-4 is also a four-character code ('MACS' for the Finder), just like the event class and event id. You might recognize this code as the Finder's creator code.

Each Macintosh file is distinguished by its file type (for example, a text file has file type "TEXT") and creator code (BBEdit's is "R*ch"). This is how the operating system knows which program to open when you double-click a desktop file. It examines the file's creator code.

Apple Event Parameters

Sometimes a lone Apple event like *quit* or *activate* will do the trick in a script. At other times, Apple events have to specify Apple event *parameters*. These are the data the receiver of the Apple event needs to carry out the Apple event's instructions. For instance, if the Example 1-3 script did not include the parameter:

```
file "mydocument"
```

then the OS 9 Finder would return an error, because its *open* Apple event requires a reference to the object(s) to open.

Example 1-3: A Finder open Command

```
tell application "Finder"
   (*open is the command; file "mydocument" is the parameter *)
   open file "mydocument"
end tell
```

Examples in this book will usually include comments explaining code elements. Comment characters in AppleScript are two hyphens (--) for single-line comments and parentheses containing asterisk characters (* *) for multi-line or single-line comments.

Figure 1-5 illustrates the Finder's *open* Apple event with the reference to the *mydocument* file.

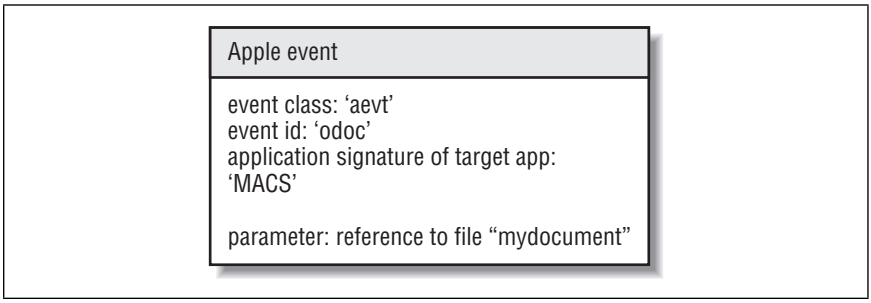

Figure 1-5: The Finder's open Apple event

Apple event parameters can include standard data types (e.g., `integer` or `string`) or references to Apple event objects, such as a document file. Apple event objects are the items or "nouns" (e.g., a file, a folder, a database record) that some scripts interact with. See the following "Apple Event Classes and Objects" section for further explanation on handling objects in your scripts.

An Apple event can have more than one required or optional parameter. In another example, if you want your script to tell FileMaker Pro to create a new row in a database, then *create* is the Apple event (followed by the required keyword `new`). The *create* Apple event requires a parameter such as a `record` object (as in a database record or row). Otherwise, how would the database program know what you wanted to create?

The code in Example 1-4 opens a database file and then creates a new record with empty fields.

Example 1-4: Getting FileMaker to Create a New Database Row

```
tell application "FileMaker Pro"
   activate --brings the target application to the front
   open file "startupdisk:fm databases:myDB.fm4"
   create new record - "record" is the parameter
end tell
```

Example 1-4 could use the *create* command's optional `with data` parameter to fill the new row with data, thus creating a complete database record.

Apple Event Classes and Objects

You have read about Apple events, which are action words or verbs (*activate*, *delete*). Apple event classes (and the objects that are based on those classes) are the nouns that your script might want to manipulate in some manner (see Table 1-3). Example 1-2 told the Finder to open a file object (basically, a file on the desktop). Objects are the data or "things" that you are interested in querying or changing when you send an Apple event to a program.

For example, a script that controls a database program usually deals with database, field, record, or cell objects. An AppleScript that sends commands to a text editor works with character, word, paragraph, and document objects.

These Apple event objects are based on classes or blueprints, such as the `file` class or the `database` class. Table 1-3 shows some of the Apple event classes from the Apple Event Registry. The operating system represents these classes internally as four-character codes.

Table 1-3: Examples of Apple Event Classes in OS 9

Class	*Four-Character Code*
`character`	`'cha '`
`disk`	`'cdis'`
`document`	`'docu'`
`file`	`'file'`
`folder`	`'cfol'`
`paragraph`	`'cpar'`
`text`	`'ctxt'`
`window`	`'cwin'`
`word`	`'cwor'`

A class is a blueprint or data type for a noun or object that you can manipulate with a script.

When an architect creates a blueprint for a structure, all the homes that are subsequently built off of the blueprint are the offspring of her original design. The real wooden, brick, or metallic homes are "instances" (in object-oriented parlance) or objects of the blueprint or class. The architect creates a `home` class in her blueprint, then the builders generate real home objects based on the original class. For example, the BBEdit text editor defines a `word` class, which is a bunch of characters that are unbroken by a space, tab, or new-line character (e.g., "apple"). The five characters that make up the word (a,p,p,l,e) are all objects based on the BBEdit `character` class. So a word object constitutes a group of character objects. If you grouped together several separate character objects they might look like ("a", "p", "p", "l", "e").

For example, when you get the Finder to open a folder with a phrase like:

```
open folder "my_folder"
```

then you are telling the Finder to open the folder object (based on the folder class) whose name is "my_folder." This line of code will specifically create a Finder window showing the contents of the folder called "my_folder."

It is always important to describe an Apple event object in your script by its containment hierarchy, which is an exact specification of where an application like the Finder can find the object. Apple event objects are located in a similar manner to taking apart one of those wooden Russian dolls, where the dolls get smaller and smaller until you finally locate the last solid peanut-shaped doll inside of all the bigger ones. In other words, if you wanted to get information about the sender of the first message in your Outlook Express inbox folder, then you couldn't just tell Outlook to:

```
get the sender of the first message
```

because the emailer would not know where to look (i.e., in the "inbox" folder) for the email message. Consider Example 1-5, which *incompletely* describes the containment hierarchy for a file (assuming that the file is not located on the desktop).

Example 1-5: A File-Access Script That Causes an Error

```
tell application "Finder"
   open file "taxes2000"
end tell
```

The Finder cannot find this file because the script does not give a complete container reference, as in:

```
open file "taxes2000" of folder "Taxes" of startup disk
```

The script will therefore produce a dialog box reporting an error if it is run.

AppleScript has a number of ways to express containment relationships.

```
file "taxes2000" of folder "Taxes" of startup disk
```

(an "inside-out" reference). This is like describing the smallest Russian doll as "the tiny doll inside the slightly bigger doll that is contained by all the larger dolls." Or, you can use a possessive form such as:

```
startup disk's (folder "Taxes"'s file "taxes2000")
```

Using the possessive style with long container references like this one is usually less readable than the inside-out method.

Elements and Properties

Two other very important characteristics of Apple event objects are *elements* and *properties*. The class that these objects are based on defines the object's elements and properties. An object has zero or more of its defined elements, and exactly one each of its properties.

For example, SoundJam™ MP, a digital music player and encoder, defines a `playlist window` class. These objects are windows that contain lists of audio tracks that can play on the computer. Figure 1-6 shows the definition of the `playlist window` class from SoundJam's dictionary. (Chapter 2 explains a program's dictionary, which can be viewed by using Script Editor's File → Open Dictionary menu command.) A `playlist window` object has three elements: `track`, `file track`, and `URL track`. Further, `playlist window`s have a modified property (a `true` or `false` value depending on whether the window was modified since it was last saved). `playlist window`s also inherit several properties from SoundJam™ MP's `window` class. So a `playlist window` object can contain zero or more "track" elements, but it only has exactly one "modified" property value.

Rest assured that it is easy to grab the values of elements and properties in AppleScript. You can use syntax such as:

```
tell app "SoundJam™ MP" to get first file track of first playlist window
```

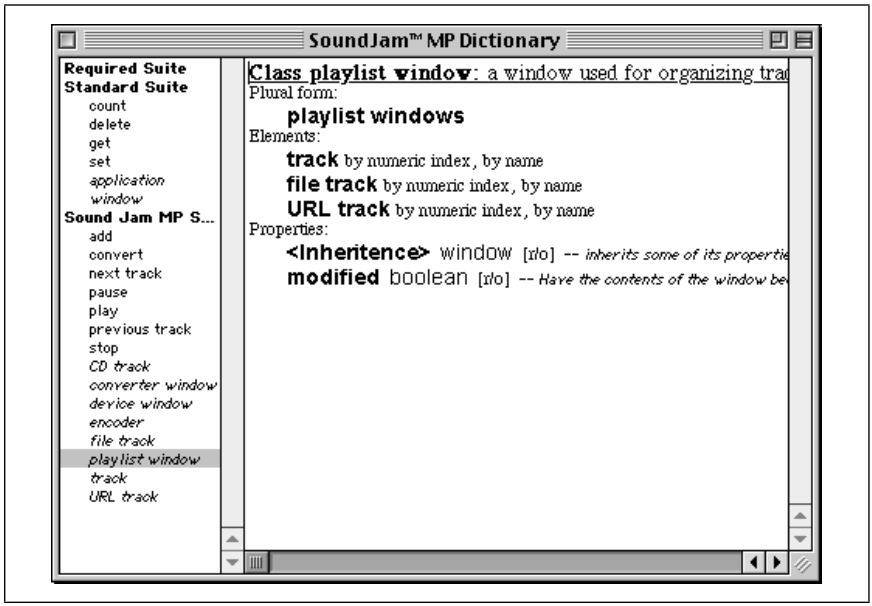

Figure 1-6: playlist window's elements/properties

This code sends SoundJam™ a *get* Apple event requesting a reference to an element, such as the first file track (an MP3 audio file) in the foremost playlist window that you see when SoundJam™ MP is open. The return value looks like:

```
'file track id 4 of playlist window id 5 of application "SoundJam™ MP"'
```

Once your script gets a reference to a track, it can then command SoundJam™ MP to play it with (you guessed it) the *play* Apple event that SoundJam™ MP defines.

Our introduction to Apple events concludes with a description of the all-important `application` class, which is the "king of the objects" in a scriptable program. The program that you script, such as application "SoundJam™ MP," is actually an object itself, an "instance" of the SoundJam™ MP `application` class.

Application Class

Many scriptable applications define an `application` class, which is the gem to study if you want to automate that program. Your quickest route to the `application` class is its description in the program's dictionary. We mentioned before that Mac programs can expose an *object model* to scripting components like AppleScript. An object model is a software abstraction, usually in tree-like form, showing the Apple event objects and Apple events that you can use with a program.

The `application` class is the root or top-level class in the program's object model. An Apple-event object model shows the `application` class and all of its elements and properties (if it has any defined elements). Figure 1-7 shows a simple object model for Sherlock 2, the Mac's fancy Find program. Sherlock 2 has three properties and contains zero or more `channel` elements. (I am sticking to

the strict definition of an object's elements, which is that an object can have zero or more of them. In reality, Sherlock 2 always has at least one defined `channel`, which is the domain that it is searching.)

`channel` elements are themselves objects with their own properties: "all search sites" and "name" (e.g., the name of one `channel` is "Internet"). When in doubt about how to script a program, always use the program's dictionary to examine its `application` class. The elements and properties of the `application` class are the things that you will be able to control and derive values from with your AppleScripts.

If you are on friendly terms with an illustration tool, then it helps to sketch out an object model of a program you are trying to script.

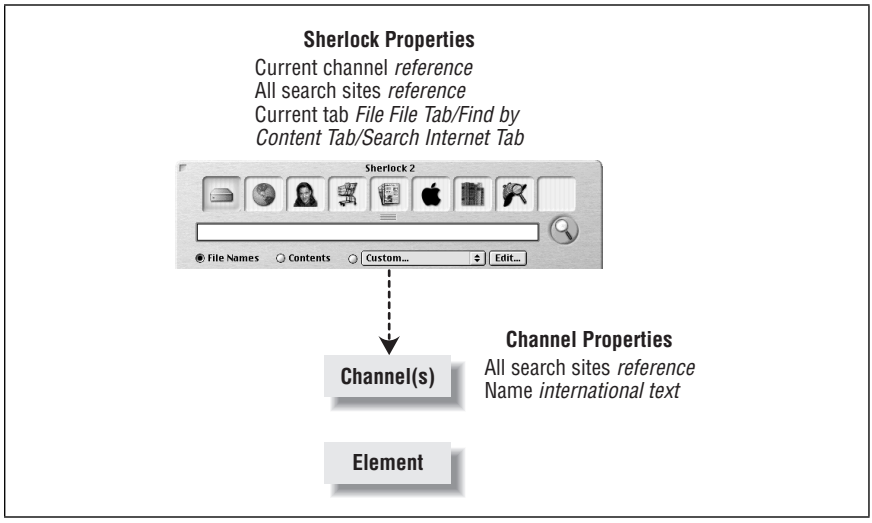

Figure 1-7: Sherlock 2's application class

Using Script Runner with OS X

OS X has a little application called Script Runner that you can use to run your scripts. Figure 1-8 shows what the open Script Runner looks like on the OS X desktop. You can find Script Runner in the AppleScript folder inside the Applications folder. Open the program by double-clicking it. If you want to add your own scripts to the Script Runner menu, choose "Open Scripts Folder" from the Script Runner menu. This opens a Finder window on to the following directory: */Users/yourname/Library/Scripts/*. Then drag any compiled scripts (they have to be saved as compiled scripts) into this window. You can of course add compiled scripts to this */Scripts* folder by navigating to it yourself (i.e., not using the Script Runner "Open Scripts Folder" menu command). After you close and restart Script Runner, you can run these scripts by choosing them in the Script Runner menu. If you create folders in the */Scripts* folder, then Script Runner will display these folders as sub-menus. This is a good way to categorize and present lots of different scripts in the Script Runner menu.

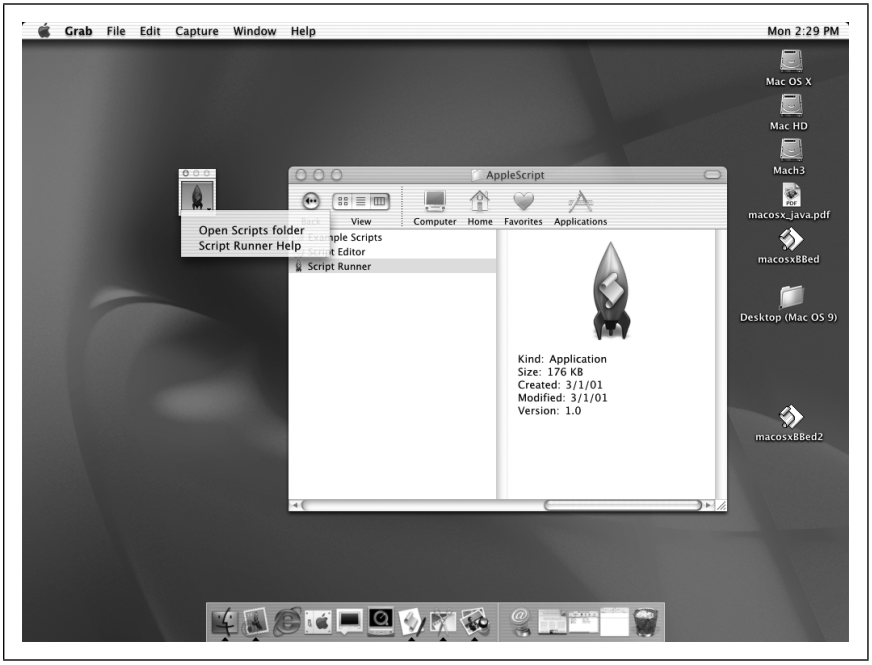

Figure 1-8: Script Runner on OS X

Using OSA Menu with OS 9

I mentioned previously that you can run AppleScripts from within Script Editor or save them as applets that can be double-clicked. In Mac OS 9, an application called OSA Menu gives you a third script running option. OSA Menu is a non-Apple system extension that you can install from the OS 9 installation CD-ROM (you can find it in the folder *CD Extras:AppleScript Extras*). OSA Menu adds its own menu to the upper-right corner of the OS 9 Finder or desktop. Figure 1-9 shows the OSA Menu after it has been activated.

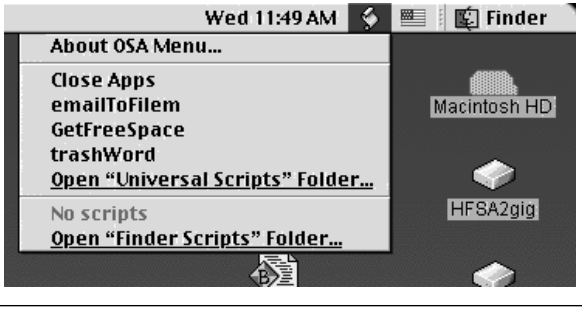

Figure 1-9: Activating the OSA Menu

The OSA Menu shows a list of compiled scripts that can be run straight from the desktop by choosing their filenames from the menu. OSA Menu will show the scripts that are stored in the following folder in OS 9: *startup disk:System Folder:Scripts:Universal Scripts*. To run properly from this menu, however, the AppleScripts have to be saved in Script Editor as compiled scripts, not applets. Chapter 2 contains more information on these script-saving options.

Checking Your AppleScript Version

This introduction would not be complete without mentioning how important it is to check which version of AppleScript is running on the machine executing your scripts. This is particularly true if your script is used on computers that might not be running Mac OS 9 or later. New versions of AppleScript are generally released along with the latest generation of the operating system. Mac OS 9 contains AppleScript Version 1.4 (an updated version of AppleScript, Version 1.4.3, also runs on OS 9). In the Spring of 2001, Mac OS 9.1 and Mac OS X used AppleScript 1.6.

There is an extremely simple way to find out which version of AppleScript is installed on the machine where the script is running. Checking the value of the version property in Script Editor will return the version number, as in `1.4` in Mac OS 9 or `1.3.4` in Mac OS 8.5. If you do not understand certain aspects of this script, Part II of the book is a detailed AppleScript language reference.

Your script can check the version property with code such as that shown in Example 1-6.

Example 1-6: Checking the AppleScript Version Number

```
set ASversion to version as string -- initialize ASversion to a string
set ASversion to (characters 1 thru 3 of ASversion as string) as real (*
coerce ASversion to a real number like 1.4 *)
if ASversion is greater than or equal to 1.4 then (* test whether the version
value is ≥ 1.4 *)
   display dialog "Good, you're running at least AppleScript 1.4" (* give the
user some feedback with a dialog box *)
else
   display dialog "You're running AppleScript " & ASversion
end if
```

Example 1-6 first gets the AppleScript version property as a `string` value (e.g., `"1.4"`) and stores it in an `ASversion` variable. The first three characters of this variable (such as `1.3` if the version was `1.3.4`) are then coerced to a `real` type, as in `1.3`. We had to take just the first three characters of the `string` because a `string` with two decimal points in it, as in `1.3.7`, cannot be coerced to a `real` value (since a `string` with two dots in it is an invalid representation of a number). Chapter 3 discusses the `real` data type.

This numerical value is then checked to see if the user is running at least AppleScript 1.4. The script uses the *display dialog* scripting addition to display information to the user about the found *version* value. You can also check the version property of the Finder, and other applications that have this property, by first targeting the application in a `tell` statement, as in Example 1-7.

Example 1-7: Displaying the Finder Version Number

```
tell application "Finder"
   set fVersion to version as string
   display dialog "You're running Finder version " & fVersion
end tell
```

Diving In

No doubt there are readers who are eager to dive into AppleScripting before they go on to this book's upcoming language reference. This section summarizes the important AppleScript language elements you need to know before you start coding:

- Case sensitivity
- Statement termination
- Line continuation character
- Naming identifiers or variables
- Variable declaration
- Comments
- Data types
- Operators and reference forms
- Flow-control statements
- Subroutines
- Script objects and libraries

All of these language elements are described in more detail in Part II (except for case sensitivity and statement termination, which are taken care of adequately in the following sections).

Case Sensitivity

Unlike other scripting languages such as JavaScript and Perl, AppleScript is not case-sensitive. In other words, `MYVAR` is the same as `myvar`, or *myfunc* is the same as *MyFunc* in terms of function definitions. Script Editor will not let you define two functions with the same name, even if their letters are different combinations of upper- and lowercase characters. The numerous AppleScript constants and reserved words (`case`, `current application`, and other constants are covered in Chapter 6, *Variables and Constants*) cannot be reused as your own variable or method names. A script can change the values of predefined variables such as `pi` or `space`; however, scripters are better off using these predefined variables for the variables' intended purpose and creating their own variable names. Script Editor sees "pi" and "PI" as the same thing ("PI" would be corrected to "pi" when you compile the script). Class and command names within applications, while mostly lowercase, are corrected when you compile the script to the spelling that is specified in the app's dictionary. (Chapter 2 explains an application's dictionary.) For instance, if you typed the class name `tcpip v4 configuration` into Script Editor, inside of a *tell app "Network Setup Scripting"* block, it would be corrected to "TCPIP v4 configuration" when the statement was compiled.

Statement Termination

You don't have to terminate an AppleScript statement using any special characters, as you do with Perl (the semi-colon character). You do, however, have to complete each statement on a line before you go on to the next statement, unless you use the continuation character (¬).

Line Continuation Character

You can split a very long statement into several lines by typing *Option-Return* on the Macintosh. This produces a continuation character (¬). This character only affects how the code looks in Script Editor and is not part of the compiled code. If you store a `string` literal in a variable, however, and add a continuation character to the middle of the literal `string`, then this character becomes a visible part of the compiled `string` of characters (you usually want to avoid this). Splitting long code statements with the continuation character makes the script more readable. You will use this character often.

Naming Identifiers or Variables and Functions

The names that you create for variables and functions have to begin with a letter or underscore character (_), but subsequent characters can include letters, numbers, and underscores. In variables or function names, you cannot include AppleScript's reserved words and operators such as *, &, ^, or + (covered in Chapter 4, *Operators*) or special characters such as $, @, or #. An exception to this AppleScript rule allows for the creation of weird variable or function names if you use vertical bars (|) to begin and end the identifier, as in: `set |2^$var| to 25`. The variable `|2^$var|` is actually valid. If you wanted to create the equivalent of a Perl scalar variable in AppleScript, you could use: `set |$perlVar| to 25`. There is no practical limit to the size of AppleScript variable names; that is, you can have a variable name that has up to 251 characters, but you would never want to deal with variable names that long. In my OS 9 testing, a variable name that exceeded 251 characters produced the error dialog in Figure 1-10.

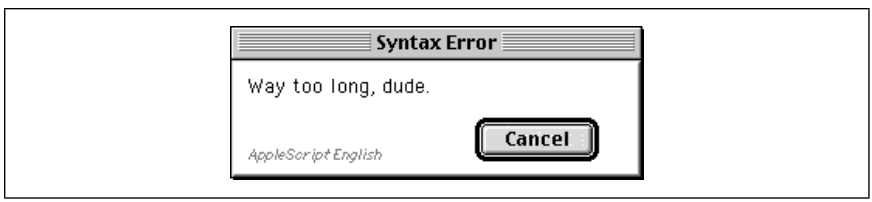

Figure 1-10: Script Editor signals a variable name that's too long

Variable Declaration

You can use either the `set` or `copy` keywords to declare a variable and assign a value to it, as in the following examples:

```
set myvar to (5 * 25)
copy (5 * 25) to myvar
```

Both of these statements produce the same result; they store the `integer` 125 in the `myvar` variable. The `set` version however is more intuitive and is used more often to declare variables. `set` and `copy` furthermore have different results when you use the variable to contain a `list`, `record`, or `script` object. (Chapter 9, *Script Objects and Libraries*, discusses this AppleScript feature.) The `copy` keyword, as in:

```
copy listVar to newListVar
```

will make a new copy of the `list` value stored in `listVar` and store this new `list` in `newListVar`. If you used:

```
set newListVar to listVar
```

the `list` stored in `newListVar` will still refer to the original `list` (i.e., `listVar`). The `newListVar` variable will not get a new copy of the `list` when you use `set`. Chapter 6 goes further into this `set` and `copy` subject.

Comments

Comments are the descriptions that you add to the code as reminders to yourself and guidance to other coders; they are not part of the executable script. AppleScript uses two or more dashes (--) preceding the comment text for single-line comments and (* *) surrounding the comment text for single- or multi-line comments. Using dashes, you can have a comment on the same line as some code, such as:

```
set myvar to 10 -- initialize myvar to 10.
```

AppleScript does not use the popular slash-slash (//) single-line comment characters of Java or C++.

Data Types

Like most scripting tools, AppleScript is a "loosely typed" programming language. This means that for the most part you do not have to specify exactly how the computer will store some data when you set a variable to a value. AppleScript takes care (or tries to) of the details for you. So when you use the code fragment: [`set num to 75`], AppleScript knows that num is an `integer` or `number`. If you use:

```
set numstr to "I'm a string"
```

`numstr` is automatically stored as a `string`. This feature does not forbid you from specifying the data type of a variable, which is a good idea in many situations and creates more readable code. If you want to explicitly set a variable to an AppleScript data type, use the `as` keyword, as in `get current date as string`. If you want to ensure that a number will be stored as a real data type, use code such as:

```
set num to 75.0
```

This code sets the variable `num` to a `real` data type, which is a very large number that can include a decimal point, similar to a `double` type in Java. A program can now increment or increase the variable `num` to a much higher number than it

could if it were left as an `integer`, which has a range of –536,870,911 to 536,870,911, inclusive. What if you wanted to have a variable keep track over time of the number of people on Earth who are connected to the Web? This number would eventually exceed one billion, so you would want to use a variable of a `real` data type.

However, explicitly setting data types in AppleScript is also a potentially error-prone strategy if you are not careful in your script planning. For example, the code: `set num to 1.5 as integer` will compile but raise an error once the script is run. The error message is "Can't make 1.5 into an integer." If you left the `as integer` part out of the code fragment, then AppleScript would automatically set num to a `real` data type and no error would occur.

The following list briefly describes the other principal data types that Apple-Scripters should be aware of (these are all covered in more detail in Chapter 3):

boolean

The literal words true or false, or an expression that evaluates to true or false. AppleScript does not treat other types of "true- or false-type" values, such as 0 or 1, as boolean values. Example 1-8 shows two ways to derive and store boolean values in AppleScript.

Example 1-8: Boolean Values in AppleScript

```
set bool to false -- bool is a boolean data type
set bool to (5 > 3)
(* bool is a boolean because the expression "(5 > 3)" evaluates to true *)
```

date

Set a variable to a date value with code such as:

```
set theDate to date "12/5/2000"
```

Remember to use the `date` keyword followed by the `date string` ("12/5/2000"), or the `theDate` variable is stored as a `string` type. A common error is to type something like `set the Date to "12/5/2000"`, which stores a `string` data type in `theDate`, not the `date` value that the scripter is aiming for. A lot of scripts get an initial date value from the useful scripting addition *current date*. When this command is used in a script, it returns a `date` object representing the current date and time, as in:

```
date "Tuesday, December 05, 2000 12:00:00 AM"
```

Appendix A, *Standard Scripting Additions,* describes the *current date* command.

String

A series of letters, spaces, numbers, or other characters delimited by double-quote marks, as in "c" or "Here is a longer string" or "" (an empty string, but a valid string nonetheless). Suffice it to say, you deal with strings all the time in AppleScript as you read data from or write data to files, database records, and other storage media. AppleScript does not allow you to define a string with single quotation marks; you have to use double quotes. Once you have a string type, then you can get its length property (an integer), which is the

number of characters that are in a string. You can use a phrase such as current date as string whose return value looks like:

```
"Tuesday, December 05, 2000 2:59:30 PM"
```

A `string` also has several elements such as words. The code fragment:

```
words of "four score and seven years ago"
```

returns a `list` type of all the words in the string (i.e., `{"four", "score", "and", "seven", "years", "ago"}`). You can concatenate two strings to make one string using the concatenation character ("&").

List

Called an array in other languages like Perl or Java. In AppleScript, you can store values of several different data types, including strings, numbers, and other lists, in the same list. Example 1-9 stores a string, a number, a list, and the pi predefined variable in the same list. You can see how incredibly useful this data type is; you will deal with lists *all* the time as an AppleScripter.

Example 1-9: AppleScript List Type

```
set myString to "A list that stores a string, a number, a list, and a ¬
constant."
set myList to {myString,75.0,{1,2,3},pi}
(* Return value of this script:
{"A list that stores a string, a number, a list, and a constant.", 75.0,
{1, 2, 3}, 3.14159265359}
*)
```

Lists are surrounded by curly braces (`{ }`), and a comma separates each list member. Variables that contain values, such as `myString` in Example 1-9, can also be stored in lists. Lists have three properties: `length`, `rest`, and `reverse`. `length` returns the number of list members, as in 4 for the list in Example 1-9. `rest` returns all the list members except for the first one, so the return value of `rest` of `myList` (from Example 1-9) would be `{75.0, {1, 2, 3}, 3.14159265359}`. Finally, `reverse` gives you the list with all of its members displayed in reverse order, as in:

```
{3.14159265359, {1, 2, 3}, 75.0, "A list that stores a string, a
number, a list, and a constant."}
```

You can obtain a member of a list by using the syntax item and the indexed position of the list member, as in:

```
item 4 of myList
```

This code returns the value 3.14159265359. Lists are one-based, meaning that the first list member is located at position 1, not 0 as in other languages' array implementations. Finally, you can concatenate or combine two lists by using the & operator. In Example 1-9, the code:

```
myList & "Another string"
```

attaches `"Another string"` to the `myList` list variable and makes it the last list member.

Record

A record consists of a series of name/value pairs separated by commas and surrounded by curly braces. Perl would call this an associative array, or Visual Basic would call it a collection. Examples are `{name: "AppleScript In a Nutshell", subject:"AppleScript"}` and `{first:"Bruce"}`. You can refer to the members of a record by the property name, as in:

```
subject of {name: "AppleScript In a Nutshell", subject:"AppleScript"}
```

This returns `"AppleScript"`. Records do not have `item` elements, so you cannot use the code item 1 of `{first:"Bruce"}`. You can change or coerce a record into a list, thus altering the data type of the value. An example is:

```
set nw to {name: "AppleScript In a Nutshell", subject: & ¬
"AppleScript"} as list
```

The return value loses all the property names from the original record: `{"AppleScript In a Nutshell", "AppleScript"}`.

Operators and Reference Forms

An operator is a symbol or token that is used with values or variables in an AppleScript expression. An example is the well-worn expression 2 + 2 = 4 (if you just dropped this expression into a Script Editor window, it would return a `boolean` value of `true`). The operators in this expression are "+" and "=". AppleScript has most of the operators that you would expect a scripting language to make available to the programmer. AppleScript also allows the scripter to use very readable English expressions for operators, such as:

```
if 5 is greater than 3 and 6 equals 6 then set bool to true
```

The principal symbolic operators are demonstrated in Example 1-10. All operators, including the English forms, are described in Chapter 4.

Example 1-10: AppleScript Operators

```
(* & concatenates one string to another, or combines two or more lists or
records *)
set twoPhrases to "One phrase " & "connected to another phrase."

(* the following code returns {"a string inside of a list", "added at the end
of a sentence."} *)
set twoLists to {"a string inside of a list"} & {"added at the end of a
sentence."}
(* & also combines two records to make one record. *)
set twoRecs to {firstn:"Amanda"} & {secondn:"Smith"}

(* parentheses and Math operators do what you would expect them to *)
set int to (5 * 6) - 8 -- returns 22

(*If you use / or ÷ the result is always a real data type. If you use div the
result is always an integer *)
set n1 to 50 / 26  -- returns 1.923076923077
set n2 to 50 ÷ 26  -- returns 1.923076923077
set n3 to 50 div 26  -- returns 1; div only returns integer data types
```